Gestão de Dependências em Projetos Go com Ordenação Topológica

Nicholas Pereira Cristófaro[1], **Nome Sobrenome do Aluno 2**[1]

[1]Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte – Minas Gerais – Brasil

nicholaspcr@gmail.com, email2@dominio.com

Resumo. *A gestão de dependências é um desafio em software moderno. Este artigo apresenta uma ferramenta que analisa e visualiza dependências em projetos Go, modelando-as como um grafo direcionado. Através da ordenação topológica com o algoritmo de Kahn, a ferramenta determina a sequência de compilação segura e detecta dependências cíclicas — um erro crítico em Go. A solução, implementada em Python, abrange desde a análise do código até a geração de um grafo interativo, oferecendo um suporte prático para a gestão de conflitos em sistemas de dependência.*

Abstract. *Dependency management is a challenge in modern software. This paper introduces a tool that analyzes and visualizes dependencies in Go projects by modeling them as a directed graph. Through topological sorting with Kahn's algorithm, the tool determines a safe compilation sequence and detects cyclic dependencies—a critical error in Go. The solution, implemented in Python, covers the process from code analysis to generating an interactive graph, providing practical support for managing conflicts in dependency systems.*

1. Introdução

O desenvolvimento de software contemporâneo é caracterizado pela modularidade e reutilização de código, resultando em sistemas compostos por um grande número de componentes interdependentes. A gestão dessa rede de dependências é um desafio central na engenharia de software. Falhas nesse gerenciamento podem levar a erros de compilação, comportamento inesperado em tempo de execução e dificuldades na manutenção e evolução dos projetos.

A linguagem de programação Go (Golang), desenvolvida pelo Google, possui um sistema de pacotes estático e um compilador que impõem regras estritas sobre as dependências. Uma dessas regras é que as dependências de um pacote devem ser inicializadas antes do próprio pacote. Isso garante a execução correta das funções 'init()', que são utilizadas para configurar o estado inicial de cada pacote. Quando um projeto contém uma dependência cíclica (por exemplo, pacote A importa B e pacote B importa A), o compilador Go gera um erro, pois não é possível determinar uma ordem de inicialização válida.

A ordem de inicialização de pacotes em Go pode ser modelada de forma natural como um grafo de dependências direcionado acíclico (DAG). Neste modelo, cada pacote é um vértice e uma diretiva 'import' de um pacote A para um pacote B cria uma aresta direcionada de B para A, significando que A depende de B. A solução para determinar a

sequência correta de inicialização é, portanto, encontrar uma ordenação topológica desse grafo.

Este trabalho se enquadra na proposta de "Gestão de Conflitos em Sistemas de Dependência", focando na modelagem, detecção de ciclos e execução segura baseada em ordenação topológica. Para isso, foi desenvolvida uma ferramenta em Python que:

- Analisa recursivamente os arquivos de um projeto Go para extrair as relações de importação.
- Constrói um grafo de dependências direcionado a partir dessas relações.
- Aplica o algoritmo de Kahn para realizar a ordenação topológica dos pacotes.
- Gera uma visualização gráfica do grafo de dependências, destacando a ordem de inicialização em camadas.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica sobre grafos, ordenação topológica e o sistema de pacotes do Go. A Seção 3 detalha a modelagem do problema e a arquitetura da ferramenta desenvolvida. A Seção 4 apresenta os experimentos realizados e discute os resultados obtidos. Finalmente, a Seção 5 conclui o trabalho e aponta direções para desenvolvimentos futuros.

2. Fundamentação Teórica

Esta seção aborda os conceitos fundamentais que sustentam o desenvolvimento deste trabalho, incluindo a teoria dos grafos, o algoritmo de ordenação topológica e as particularidades do sistema de pacotes da linguagem Go.

2.1. Teoria dos Grafos

Um grafo $G = (V, E)$ é uma estrutura matemática usada para modelar relações entre objetos. Consiste em um conjunto de vértices V (ou nós) e um conjunto de arestas E que conectam pares de vértices. Em um **grafo direcionado** (ou digrafo), as arestas têm uma direção, sendo representadas por pares ordenados de vértices (u, v), indicando uma ligação de u para v.

No contexto deste trabalho, o modelo utilizado é um grafo direcionado, onde:

- **Vértices (V):** Representam os pacotes do projeto Go, tanto os locais (definidos no projeto) quanto os externos (bibliotecas padrão ou de terceiros).
- **Arestas (E):** Uma aresta direcionada (u, v) existe se o pacote v importa o pacote u. Isso significa que u é uma dependência de v.

Uma métrica importante em grafos direcionados é o **grau de entrada (in-degree)** de um vértice, que corresponde ao número de arestas que chegam a ele. Um vértice com grau de entrada zero não possui dependências dentro do grafo.

2.2. Ordenação Topológica

A ordenação topológica de um grafo direcionado é uma ordenação linear de seus vértices tal que para toda aresta direcionada (u, v), o vértice u vem antes de v na ordenação. Uma ordenação topológica só é possível se, e somente se, o grafo não contiver ciclos direcionados, ou seja, se for um **Grafo Acíclico Direcionado (DAG)**.

A existência de uma ordenação topológica no nosso grafo de dependências corresponde a uma sequência válida de inicialização dos pacotes Go. Se o grafo contiver um ciclo, não há ordenação topológica possível, o que corresponde a um erro de "dependência cíclica" no compilador Go.

2.2.1. Algoritmo de Kahn

Existem vários algoritmos para encontrar uma ordenação topológica. O escolhido para este trabalho foi o **algoritmo de Kahn**, devido à sua capacidade de processar os vértices em ”camadas”, o que é ideal para a visualização proposta. O algoritmo funciona da seguinte maneira:

1. Calcular o grau de entrada (in-degree) de todos os vértices do grafo.
2. Inicializar uma fila com todos os vértices que possuem grau de entrada igual a zero. Estes são os pacotes sem dependências.
3. Enquanto a fila não estiver vazia:
   (a) Remover um vértice n da fila e adicioná-lo à lista de ordenação topológica.
   (b) Para cada vértice m que é vizinho de n (ou seja, para cada pacote que depende de n):
      i. Decrementar o grau de entrada de m.
      ii. Se o grau de entrada de m se tornar zero, adicioná-lo à fila.
4. Ao final, se o número de vértices na lista de ordenação topológica for igual ao número total de vértices do grafo, a ordenação foi bem-sucedida. Caso contrário, o grafo contém pelo menos um ciclo.

A implementação deste algoritmo pode ser vista no arquivo ‘topological_sorting.py‘.

2.3. Sistema de Pacotes e Módulos em Go

Go organiza o código em pacotes, que são coleções de arquivos-fonte no mesmo diretório. O ‘go.mod‘ é um arquivo na raiz do projeto que define o caminho do módulo, que serve como um prefixo para todos os pacotes dentro do projeto, e também gerencia as dependências externas.

A diretiva ‘import‘ é usada para declarar dependências entre pacotes. Como mencionado, a ordem de execução das funções ‘init()‘ de cada pacote segue a ordem de dependência, exigindo uma resolução que é, na prática, uma ordenação topológica.

3. Modelagem e Desenvolvimento da Ferramenta

Esta seção descreve a arquitetura e os componentes da ferramenta desenvolvida, detalhando como o problema de análise de dependências foi modelado e implementado em Python. A ferramenta é composta por três módulos principais, conforme os arquivos ‘dependency_graph_builder.py‘, ‘topological_sorting.py‘ e ‘render.py‘.

3.1. Modelagem do Grafo de Dependências

O problema foi modelado como um grafo direcionado $G = (V, E)$, conforme descrito na Seção 2.1. A estrutura de dados escolhida para representar o grafo foi um dicionário (hash map) em Python, onde as chaves são os nomes dos pacotes (vértices) e os valores são conjuntos contendo os pacotes que dependem deles (vizinhos).

- **‘graph = defaultdict(set)‘:** Representa as arestas. ‘graph[u]‘ contém um conjunto de vértices ‘v1, v2, ...‘ para os quais existe uma aresta $(u, v1)$, $(u, v2)$, etc.
- **‘in_degree = defaultdict(int)‘:** Armazena o grau de entrada de cada vértice, essencial para o algoritmo de Kahn.

3.2. Componentes da Ferramenta

A ferramenta segue um fluxo de execução claro: extração de dados, processamento (ordenação) e visualização.

3.2.1. Extração e Construção do Grafo (`dependency_graph_builder.py`)

Este módulo é responsável por ler o sistema de arquivos e construir a representação do grafo. O processo é o seguinte:

1. **Identificação do Módulo:** A classe ‘BuildDependencyGraph‘ inicia lendo o arquivo ‘go.mod‘ para determinar o prefixo do módulo do projeto. Isso é crucial para nomear corretamente os pacotes locais.
2. **Busca por Arquivos Go:** O método ‘find_go_files‘ percorre recursivamente o diretório do projeto em busca de arquivos com a extensão ‘.go‘, ignorando arquivos de teste (‘_test.go‘).
3. **Extração de Importações:** O método estático ‘extract_imports‘ utiliza expressões regulares (regex) para analisar o conteúdo de cada arquivo Go e extrair todos os pacotes importados. Ele é projetado para lidar tanto com importações de linha única (‘import ”fmt”‘) quanto com blocos de importação (‘import (...)‘).
4. **Construção do Grafo:** O método ‘build_dependency_graph‘ orquestra o processo. Ele primeiro identifica todos os pacotes definidos localmente e depois itera sobre cada arquivo, usando as importações extraídas para construir o ‘graph‘ e popular o mapa ‘in_degree‘.

```
import (
    "fmt"
    "log"
    "net/http"

    "github.com/gorilla/mux"
    "meuprojeto/internal/database"
)
```

Listing 1. Exemplo de um bloco de importações em Go.

```
import "fmt"
```

Listing 2. Exemplo de importação inline em Go.

3.2.2. Ordenação Topológica e Detecção de Ciclos (`topological_sorting.py`)

Este módulo implementa a lógica central do processamento.

- **Detecção de Ciclos:** Antes de tentar a ordenação, o método ‘is_cyclic‘ é chamado. Ele utiliza um algoritmo de busca em profundidade (DFS) para verificar a existência de ciclos no grafo. Se um ciclo é detectado, o programa informa o usuário e encerra, pois a ordenação topológica é impossível.
- **Ordenação com Algoritmo de Kahn:** O método ‘sort_group‘ implementa o algoritmo de Kahn, conforme descrito na Seção 2.2.1. Em vez de retornar uma lista linear, nossa implementação retorna uma lista de listas (‘list[list[str]]‘), onde

cada lista interna representa uma ”camada” de pacotes que podem ser processados/compilados em paralelo, pois suas dependências da camada anterior já foram resolvidas.

3.2.3. Visualização dos Resultados (`render.py`)

Para facilitar a compreensão da estrutura de dependências, um módulo de visualização foi criado utilizando a biblioteca ‘graphviz‘.

1. O grafo é renderizado como um ‘Digraph‘ com orientação da esquerda para a direita (‘rankdir=’LR’‘).
2. Os nós são agrupados em subgrafos com o mesmo ‘rank‘, correspondendo às camadas calculadas pela função ‘sort_group‘. Isso alinha visualmente os pacotes que estão no mesmo nível da ordenação topológica.
3. Os nós são coloridos de forma distinta: pacotes locais (parte do projeto analisado) têm uma cor, e pacotes externos (bibliotecas padrão ou de terceiros) têm outra. Isso ajuda a diferenciar o código do projeto de suas dependências externas.
4. O resultado final é um arquivo SVG embutido em um HTML, que é automaticamente aberto no navegador, proporcionando uma visualização limpa e interativa.

4. Experimentos e Resultados

Para validar a ferramenta, foi realizado um experimento com um projeto Go hipotético, cuja estrutura de dependências é complexa o suficiente para demonstrar a eficácia da abordagem.

4.1. Cenário de Teste

Consideremos um projeto hipotético de um e-commerce com a seguinte estrutura de pacotes e dependências:

- **‘main‘:** O ponto de entrada da aplicação. Depende de ‘api‘ e ‘config‘.
- **‘api‘:** Define os handlers da API. Depende de ‘services‘.
- **‘services‘:** Contém a lógica de negócio. Depende de ‘repository‘ e ‘models‘.
- **‘repository‘:** Camada de acesso a dados. Depende de ‘models‘ e de um pacote externo, ‘github.com/jmoiron/sqlx‘.
- **‘models‘:** Define as estruturas de dados. Não possui dependências internas.
- **‘config‘:** Gerencia as configurações. Depende de um pacote padrão, ‘os‘.

4.2. Execução da Ferramenta e Análise dos Resultados

Ao executar a ferramenta no diretório deste projeto, a saída da ordenação topológica em camadas, calculada pelo ‘sort_group‘, seria a seguinte:

```
[
  ["os", "github.com/jmoiron/sqlx", "models"],
  ["config", "repository"],
  ["services"],
  ["api"],
  ["main"]
]
```

Esta saída demonstra a ordem de inicialização correta:

1. **Camada 0:** Os pacotes ‘os‘, ‘github.com/jmoiron/sqlx‘ e ‘models‘ não têm dependências internas no projeto e podem ser inicializados primeiro.
2. **Camada 1:** ‘config‘ (que depende de ‘os‘) e ‘repository‘ (que depende de ‘models‘ e ‘sqlx‘) são os próximos.
3. **Camada 2:** ‘services‘ pode ser inicializado, pois suas dependências (‘repository‘ e ‘models‘) já estão resolvidas.
4. **Camada 3 e 4:** ‘api‘ e, finalmente, ‘main‘ são inicializados em sequência.

O grafo visual gerado pelo ‘render.py‘ reflete essa estrutura, como pode ser visto na Figura 1.

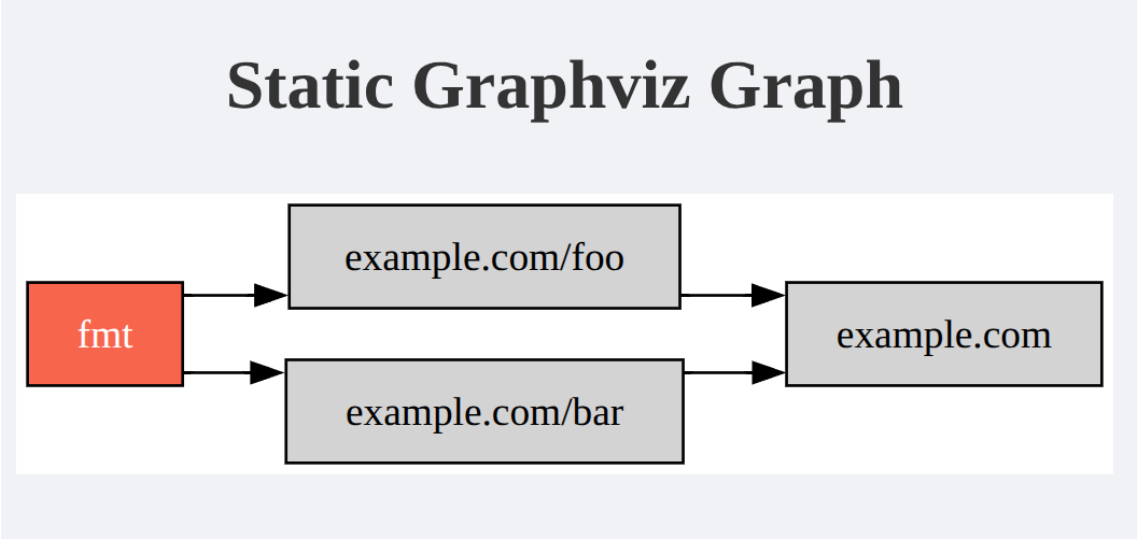

Figure 1. Visualização do grafo de dependências gerado para o exemplo simples. Os nós em vermelho representam pacotes externos/padrão.

A clareza da visualização (Figura 1) e a ordem explícita dos pacotes fornecem ao desenvolvedor uma visão imediata da arquitetura do projeto, facilitando a depuração de problemas de inicialização e a compreensão do fluxo de compilação.

5. Conclusão

Este trabalho apresentou uma solução computacional para o problema de gestão de dependências em projetos Go, alinhado aos objetivos da disciplina de Ferramenta de Processamento de Dados com Grafos. A modelagem do sistema de pacotes como um grafo direcionado provou-se eficaz, e a aplicação da ordenação topológica através do algoritmo de Kahn permitiu não apenas determinar a sequência de compilação segura, mas também estruturar a visualização dos resultados de forma intuitiva.

A ferramenta desenvolvida cumpre com sucesso os requisitos propostos: ela analisa o código-fonte, constrói um modelo em grafo, processa-o com algoritmos relevantes e apresenta os resultados de forma clara. A capacidade de detectar ciclos de dependência antes da compilação é um recurso valioso para evitar erros comuns em projetos de grande porte.

5.1. Trabalhos Futuros

Apesar de funcional, a ferramenta atual pode ser estendida de várias maneiras. Como trabalhos futuros, sugere-se:

- **Análise de Desempenho:** Testar a ferramenta em repositórios de código aberto extremamente grandes (e.g., Kubernetes, Docker) para avaliar sua performance e escalabilidade.
- **Parser Avançado:** Substituir as expressões regulares por um parser de Abstract Syntax Tree (AST) da linguagem Go, o que tornaria a extração de importações mais robusta e precisa.
- **Integração com IDEs:** Desenvolver um plugin para editores de código populares, como o Visual Studio Code, que exiba o grafo de dependências e alerte sobre ciclos em tempo real.
- **Análise de "Peso" das Arestas:** Estender o modelo para incluir métricas nas arestas, como o número de funções ou tipos utilizados de um pacote dependente, para identificar acoplamentos fortes.

References

Boulic, R. and Renault, O. (1991). 3d hierarchies for animation. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd.

Knuth, D. E. (1984). *The TEX Book*. Addison-Wesley, 15th edition.

Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.